5. 병렬 처리

스레드 시작과 정지

```
from threading import Thread
import time

class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print("T-minus", n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
```

"코드를 병렬적으로 실행하기 위해서 스레드를 만들거나 없애고 싶다."

```
time.sleep(20)
print('About to terminate')
c.terminate()
t.join()
print('Terminated')
```

```
T-minus 10
T-minus 9
T-minus 8
T-minus 7
About to terminate
Terminated
```

"threading 라이브러리는 파이썬의 호출 가능한 것을 스스로 스레드에서 실행하도록 한다. Thread 인스턴스를 만들고 실행하고 싶은 것을 넣으면 된다."

스레드가 시작했는지 판단하기

```
from threading import Thread, Event
import time

def countdown(n, started_evt):
    print("countdown starting")
    started_evt.set()
    while n > 0:
        print("T-minus", n)
        n -= 1
        time.sleep(5)

started_evt = Event()

print("Launching countdown")
t = Thread(target=countdown, args=(10,started_evt))
t.start()
```

"스레드를 만들었는데, 실제로 실행을 시작했는지 확인하고 싶다."

스레드가 시작했는지 판단하기

```
started_evt.wait()
print("countdown is running")
```

```
Launching countdown
countdown starting
T-minuscountdown is running 10

T-minus 9
T-minus 8
T-minus 7
T-minus 6
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
```

"스레드가 특정 지점에 도달했는지 알아야 하는 경우에 복잡한 동기화 문제가 생기기도 한다. 이 문제를 해결하려면 threading 라이브러리의 Event 객체를 사용한다."

스레드가 시작했는지 판단하기


```
import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
        t.daemon = True
        t.start()

    def run(self):
        while True:
            time.sleep(self._interval)
            with self._cv:
                self._flag ^= 1
                self._cv.notify_all()
```

스레드가 시작했는지 판단하기

```
    def wait_for_tick(self):
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

ptimer = PeriodicTimer(5)
ptimer.start()

def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print("T-minus", nticks)
        nticks -= 1
```

스레드가 시작했는지 판단하기


```
def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print("Counting", n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()
```

```
T-minusCounting 0
 10
CountingT-minus 9
 1
T-minusCounting  2
8
CountingT-minus  3
7
CountingT-minus 6
 4
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
```

스레드가 시작했는지 판단하기

```
import threading
import time

def worker(n, sema):
    sema.acquire()
    print("Working", n)

sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.daemon=True
    t.start()
```

스레드가 시작했는지 판단하기

```
print('About to release first worker')
time.sleep(5)
sema.release()
time.sleep(1)
print('About to release second worker')
time.sleep(5)
sema.release()
time.sleep(1)
print('Goodbye')
```

```
About to release first worker
Working 0
About to release second worker
Working 1
Goodbye
```

```
from queue import Queue
from threading import Thread
import time

_sentinel = object()

def producer(out_q):
    n = 10
    while n > 0:
        out_q.put(n)
        time.sleep(2)
        n -= 1
    out_q.put(_sentinel)

def consumer(in_q):
    while True:
        data = in_q.get()

        if data is _sentinel:
            in_q.put(_sentinel)
            break

        print('Got:', data)
    print('Consumer shutting down')
```

```
if __name__ == '__main__':
    q = Queue()
    t1 = Thread(target=consumer, args=(q,))
    t2 = Thread(target=producer, args=(q,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

```
Got: 10
Got: 9
Got: 8
Got: 7
Got: 6
Got: 5
Got: 4
Got: 3
Got: 2
Got: 1
Consumer shutting down
```

"프로그램에서 다중 스레드를 사용하는 중이고, 스레드끼리 안전하게 통신하거나 데이터를 주고받게 만들고 싶다."

```
import heapq
import threading
import time

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

```
def producer(q):
    print('Producing items')
    q.put('C', 5)
    q.put('A', 15)
    q.put('B', 10)
    q.put('D', 0)
    q.put(None, -100)

def consumer(q):
    time.sleep(5)
    print('Getting items')
    while True:
        item = q.get()
        if item is None:
            break
        print('Got:', item)
    print('Consumer done')
```

```
if __name__ == '__main__':
    q = PriorityQueue()
    t1 = threading.Thread(target=producer, args=(q,))
    t2 = threading.Thread(target=consumer, args=(q,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

```
Producing items
Getting items
Got: A
Got: B
Got: C
Got: D
Consumer done
```

"우선순위 큐를 스레드에 안전하게 사용하는 코드 구현."

5. 병렬 처리

```
import threading

class SharedCounter:
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        with self._value_lock:
             self._value += delta

    def decr(self,delta=1):
        with self._value_lock:
             self._value -= delta

def test(c):
    for n in range(1000000):
        c.incr()
    for n in range(1000000):
        c.decr()
```

임계 영역 락

```
if __name__ == '__main__':
    c = SharedCounter()
    t1 = threading.Thread(target=test, args=(c,))
    t2 = threading.Thread(target=test, args=(c,))
    t3 = threading.Thread(target=test, args=(c,))
    t1.start()
    t2.start()
    t3.start()
    print('Running test')
    t1.join()
    t2.join()
    t3.join()

    assert c._value == 0
    print('Looks good!')
```

```
Running test
Looks good!
```

"프로그램이 스레드를 사용하고 코드의 임계 영역을 락해서 레이스 컨디션 상황을 피하고 싶다."

락킹으로 데드락 피하기


```
import threading
from contextlib import contextmanager

_local = threading.local()

@contextmanager
def acquire(*locks):
    locks = sorted(locks, key=lambda x: id(x))

    acquired = getattr(_local, 'acquired',[])
    if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
        raise RuntimeError('Lock Order Violation')

    acquired.extend(locks)
    _local.acquired = acquired
    try:
        for lock in locks:
            lock.acquire()
        yield
    finally:
        for lock in reversed(locks):
            lock.release()
        del acquired[-len(locks):]
```

락킹으로 데드락 피하기


```
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print("Thread-1")

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print("Thread-2")

input('This program runs forever. Press [return] to start, Ctrl-C to exit')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()

import time
while True:
    time.sleep(1)
```

락킹으로 데드락 피하기


```
import threading
import time

x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock):
            with acquire(y_lock):
                print("Thread-1")
                time.sleep(1)

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print("Thread-2")
                time.sleep(1)

input('This program crashes with an exception. Press [return] to start')
```

락킹으로 데드락 피하기

```
t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()

time.sleep(5)
```

```
import threading

def philosopher(left, right):
    while True:
        with acquire(left,right):
             print(threading.currentThread(), 'eating')

NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                         args=(chopsticks[n],chopsticks[(n+1) % NSTICKS]))
    t.daemon = True
    t.start()

import time
while True:
    time.sleep(1)
```

특정 스레드 용 상태 저장

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        if hasattr(self.local, 'sock'):
            raise RuntimeError('Already connected')
        self.local.sock = socket(self.family, self.type)
        self.local.sock.connect(self.address)
        return self.local.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.sock.close()
        del self.local.sock
```

"다른 스레드에는 보이지 않고 현재 실행 중인 스레드에만 사용할 상태를 저장하고 싶다."

특정 스레드 용 상태 저장

```
def test(conn):
    from functools import partial

    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')
        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got {} bytes'.format(len(resp)))

if __name__ == '__main__':
    conn = LazyConnection(('www.python.org', 80))

    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

"스레드에만 사용할 데이터를 저장해야 할 상황이 있다면 threading.local()을 사용해서 스레드-로컬-저장소 객체를 만들어야 한다."

특정 스레드 용 상태 저장

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        sock = socket(self.family, self.type)
        sock.connect(self.address)
        if not hasattr(self.local, 'connections'):
            self.local.connections = []
        self.local.connections.append(sock)
        return sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.connections.pop().close()
```

특정 스레드 용 상태 저장

```
def test(conn):
    from functools import partial

    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')
        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got {} bytes'.format(len(resp)))

    with conn as s1, conn as s2:
        s1.send(b'GET /downloads HTTP/1.0\r\n')
        s2.send(b'GET /index.html HTTP/1.0\r\n')
        s1.send(b'Host: www.python.org\r\n')
        s2.send(b'Host: www.python.org\r\n')
        s1.send(b'\r\n')
        s2.send(b'\r\n')
        resp1 = b''.join(iter(partial(s1.recv, 8192), b''))
        resp2 = b''.join(iter(partial(s2.recv, 8192), b''))

    print('resp1 got {} bytes'.format(len(resp1)))
    print('resp2 got {} bytes'.format(len(resp2)))
```

특정 스레드 용 상태 저장

```
if __name__ == '__main__':

    conn = LazyConnection(('www.python.org', 80))
    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t3 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t3.start()
    t1.join()
    t2.join()
    t3.join()
```

5. 병렬 처리

5.1 병렬 처리 I

5.2 병렬 처리 II

5.3 병렬 처리 III

5.4 병렬 처리 IV

스레드 풀 생성

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()
```

"워크 스레드 풀을 만들어 클라이언트를 처리하거나 기타 동작을 하고 싶다."

스레드 풀 생성

```
def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

```
Echo server running at ('', 15000)
```

"concurrent.futures 라이브러리에 ThreadPoolExecutor 클래스로 해결한다."

스레드 풀 생성

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()
```

"스레드 풀을 스스로 만들고 싶다면 Queue를 사용하면 간단하다."

스레드 풀 생성

```
def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

스레드 풀 생성

```
from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

x = a.result()
y = b.result()
```

"스레드 풀을 직접 구현하지 않고 ThreadPoolExecutor를 사용하면 호출한 함수에서 그 결과를 더 쉽게 받을 수 있다는 장점이 있다.

a.result()는 일치하는 함수가 풀에 의해 실행되고 값을 반환할 때까지 실행을 멈춘다."

간단한 병렬 프로그램 수행


```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

간단한 병렬 프로그램 수행

```
'http://www.foxnews.com/' page is 330194 bytes
'http://www.cnn.com/' page is 1130810 bytes
'http://some-made-up-domain.com/' page is 64668 bytes
'http://www.bbc.co.uk/' page is 278708 bytes
```

5. 병렬 처리

5.1 병렬 처리 I

5.2 병렬 처리 II

5.3 병렬 처리 III

5.4 병렬 처리 IV

액터 작업 정의

```
from queue import Queue
from threading import Thread, Event

class ActorExit(Exception):
    pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        self._mailbox.put(msg)

    def recv(self):
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        self.send(ActorExit)
```

액터 작업 정의

```
def start(self):
        self._terminated = Event()
        t = Thread(target=self._bootstrap)
        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
        self._terminated.wait()

    def run(self):
        while True:
            msg = self.recv()
```

액터 작업 정의

```
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print("Got:", msg)

if __name__ == '__main__':
    p = PrintActor()
    p.start()
    p.send("Hello")
    p.send("World")
    p.close()
    p.join()
```

```
Got: Hello
Got: World
```

액터 작업 정의

```
class TaggedActor(Actor):
    def run(self):
        while True:
            tag, *payload = self.recv()
            getattr(self,"do_"+tag)(*payload)

    def do_A(self, x):
        print("Running A", x)

    def do_B(self, x, y):
        print("Running B", x, y)

if __name__ == '__main__':
    a = TaggedActor()
    a.start()
    a.send(('A', 1))
    a.send(('B', 2, 3))
    a.close()
    a.join()
```

```
Running A 1
Running B 2 3
```

액터 작업 정의

```
from threading import Event

class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value
        self._evt.set()

    def result(self):
        self._evt.wait()
        return self._result
```

액터 작업 정의

```
class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

if __name__ == '__main__':
    worker = Worker()
    worker.start()
    r = worker.submit(pow, 2, 3)
    print(r.result())
    worker.close()
    worker.join()
```

```
8
```

메시지 출판/구독 구현

```
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

_exchanges = defaultdict(Exchange)

def get_exchange(name):
    return _exchanges[name]
```

```
if __name__ == '__main__':
    class Task:
        def __init__(self, name):
            self.name = name
        def send(self, msg):
            print('{} got: {!r}'.format(self.name, msg))

    task_a = Task('A')
    task_b = Task('B')

    exc = get_exchange('spam')
    exc.attach(task_a)
    exc.attach(task_b)
    exc.send('msg1')
    exc.send('msg2')

    exc.detach(task_a)
    exc.detach(task_b)
    exc.send('msg3')
```

```
A got: 'msg1'
B got: 'msg1'
A got: 'msg2'
B got: 'msg2'
```

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
        for task in tasks:
            self.attach(task)
        try:
            yield
        finally:
            for task in tasks:
                self.detach(task)
```

```
 def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

_exchanges = defaultdict(Exchange)

def get_exchange(name):
    return _exchanges[name]

if __name__ == '__main__':
    class Task:
        def __init__(self, name):
            self.name = name
        def send(self, msg):
            print('{} got: {!r}'.format(self.name, msg))

    task_a = Task('A')
    task_b = Task('B')
```

```
    exc = get_exchange('spam')
    with exc.subscribe(task_a, task_b):
        exc.send('msg1')
        exc.send('msg2')

    exc.send('msg3')
```

```
A got: 'msg1'
B got: 'msg1'
A got: 'msg2'
B got: 'msg2'
```

스레드의 대안으로 제너레이터 사용


```
def countdown(n):
    while n > 0:
        print("T-minus", n)
        yield
        n -= 1
    print("Blastoff!")

def countup(n):
    x = 0
    while x < n:
        print("Counting up", x)
        yield
        x += 1

from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        self._task_queue.append(task)
```

스레드의 대안으로 제너레이터 사용

```
 def run(self):
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                next(task)
                self._task_queue.append(task)
            except StopIteration:
                pass

sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

스레드의 대안으로 제너레이터 사용


```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = { }
        self._msg_queue = deque()

    def new_actor(self, name, actor):
        self._msg_queue.append((actor,None))
        self._actors[name] = actor

    def send(self, name, msg):
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor,msg))

    def run(self):
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                 actor.send(msg)
            except StopIteration:
                 pass
```

스레드의 대안으로 제너레이터 사용

```
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            n = yield
            if n == 0:
                break
            sched.send('printer', n)
            sched.send('counter', n-1)

    sched = ActorScheduler()
    sched.new_actor('printer', printer())
    sched.new_actor('counter', counter(sched))

    sched.send('counter', 10000)
    sched.run()
```

다중 스레드 큐 폴링

```
import queue
import socket
import os

class PollableQueue(queue.Queue):
    def __init__(self):
        super().__init__()
        if os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self._putsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()

    def fileno(self):
        return self._getsocket.fileno()

    def put(self, item):
        super().put(item)
        self._putsocket.send(b'x')
```

다중 스레드 큐 폴링

```
 def get(self):
        self._getsocket.recv(1)
        return super().get()

if __name__ == '__main__':
    import select
    import threading
    import time

    def consumer(queues):
        while True:
            can_read, _, _ = select.select(queues,[],[])
            for r in can_read:
                item = r.get()
                print('Got:', item)

    q1 = PollableQueue()
    q2 = PollableQueue()
    q3 = PollableQueue()
    t = threading.Thread(target=consumer, args=([q1,q2,q3],))
    t.daemon = True
    t.start()
```

```
    q1.put(1)
    q2.put(10)
    q3.put('hello')
    q2.put(15)

    time.sleep(1)
```

```
Got: 1
Got: 10
Got: hello
Got: 15
```